高大上的淘宝架构

我们以淘宝架构为例，了解下大型的电商项目的服务端的架构是怎样，如下图所示：



上面是一些安全体系系统，如数据安全体系、应用安全体系、前端安全体系等。

中间是业务运营服务系统，如会员服务、商品服务、店铺服务、交易服务等。

还有共享业务，如分布式数据层、数据分析服务、配置服务、数据搜索服务等。

最下面呢，是中间件服务，如MQS即队列服务，OCS即缓存服务等。

图中也有一些看不到，例如高可用的一个体现，实现双机房容灾和异地机房单元化部署，为淘宝业务提供稳定、高效和易于维护的基础架构支撑。

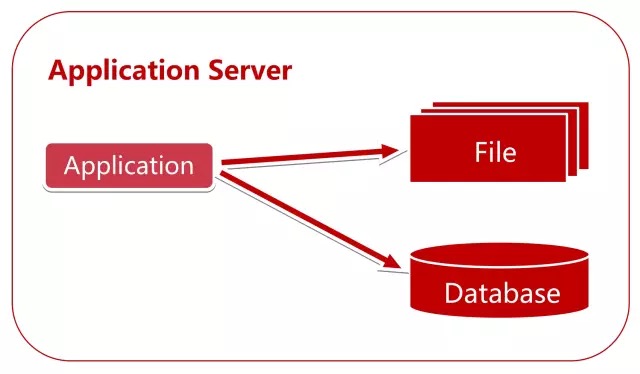
这是一个含金量非常高的架构，也是一个非常复杂而庞大的架构。当然这个也不是一天两天演进成这样的，也不是一上来就设计并开发成这样高大上的架构的。

这边就要说一下，小型公司要怎么做呢？对很多创业公司而言，很难在初期就预估到流量十倍、百倍以及千倍以后网站架构会是什么样的一个状况。同时，如果系统初期就设计一个千万级并发的流量架构，很难有公司可以支撑这个成本。

因此，一个大型服务系统都是从小一步一步走过来的，在每个阶段，找到对应该阶段网站架构所面临的问题，然后在不断解决这些问题，在这个过程中整个架构会一直演进。

那我们来一起看一下。

1. 单服务器-俗称all in one

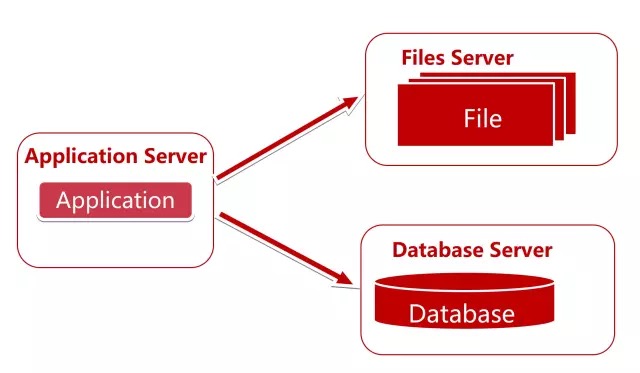


从一个小网站说起。一台服务器也就足够了。文件服务器，数据库，还有应用都部署在一台机器，俗称ALL IN ONE。

随着我们用户越来越多，访问越来越大，硬盘，CPU，内存等都开始吃紧，一台服务器已经满足不了。

这个时候看一下下一步演进。

2. 数据服务与应用服务分离

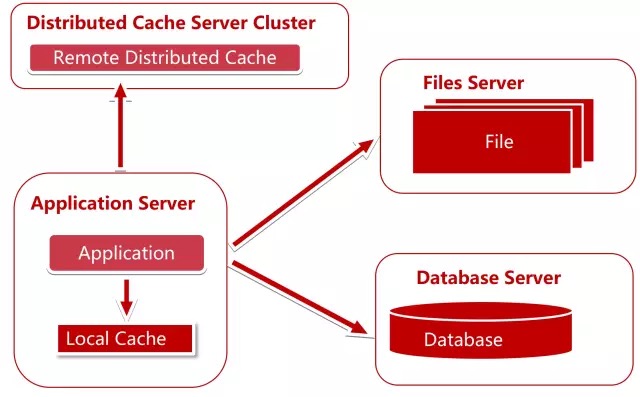


我们将数据服务和应用服务分离，给应用服务器配置更好的 CPU，内存。而给数据服务器配置更好更大的硬盘。

分离之后提高一定的可用性，例如Files Server挂了，我们还是可以操作应用和数据库等。

随着访问qps越来越高，降低接口访问时间，提高服务性能和并发，成为了我们下一个目标，发现有很多业务数据不需要每次都从数据库获取。

3. 使用缓存，包括本地缓存，远程缓存，远程分布式缓存



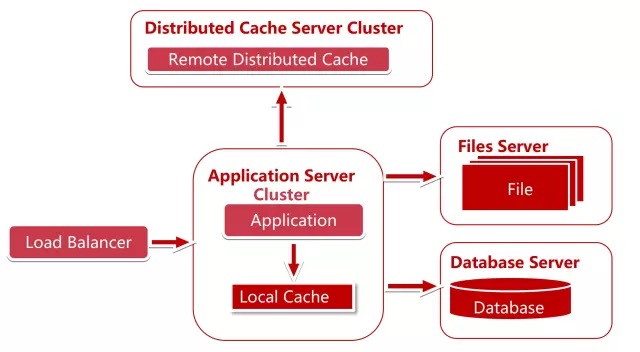
因为 80% 的业务访问都集中在 20% 的数据上，也就是我们经常说的28法则。如果我们能将这部分数据缓存下来，性能一下子就上来了。而缓存又分为两种：本地缓存和远程缓存缓存，以及远程分布式缓存，我们这里面的远程缓存图上画的是分布式的缓存集群(Cluster)。

3.1 思考的点

* 具有哪种业务特点数据使用缓存？
* 具有哪种业务特点的数据使用本地缓存？
* 具有哪种务特点的数据使用远程缓存？
* 分布式缓存在扩容时候会碰到什么问题？如何解决？分布式缓存的算法都有哪几种？各有什么优缺点？

这个时候随着访问qps的提高，服务器的处理能力会成为瓶颈。虽然是可以通过购买更强大的硬件，但总会有上限，而且这个到后期成本就是指数级增长了，这时，我们就需要服务器的集群。需要使我们的服务器可以横向扩展，这时，就必须加个新东西：负载均衡调度服务器。

4. 使用负载均衡，进行服务器集群



增加了负载均衡，服务器集群之后，我们可以横向扩展服务器，解决了服务器处理能力的瓶颈。

4.1 思考的点

* 负载均衡的调度策略都有哪些？
* 各有什么优缺点？
* 各适合什么场景？

打个比方，我们有轮询，权重，地址散列，地址散列又分为原ip地址散列hash，目标ip地址散列hash，最少连接，加权最少连接，还有继续升级的很多种策略……

我们一起来分析一下。

典型负载均衡策略分析

* 轮询：优点：实现简单，缺点：不考虑每台服务器处理能力；
* 权重：优点：考虑了服务器处理能力的不同；
* 地址散列：优点：能实现同一个用户访问同一个服务器；
* 最少连接：优点：使集群中各个服务器负载更加均匀；
* 加权最少连接：在最少连接的基础上，为每台服务器加上权值。算法为(活动连接数\*256+非活动连接数)/权重，计算出来的值小的服务器优先被选择。

继续引出问题的场景

我们的登录的时候登录了A服务器，session信息存储到A服务器上了，假设我们使用的负载均衡策略是ip hash，那么登录信息还可以从A服务器上访问，但是这个有可能造成某些服务器压力过大，某些服务器又没有什么压力，这个时候压力过大的机器(包括网卡带宽)有可能成为瓶颈，并且请求不够分散。

这时候我们使用轮询或者最小连接负载均衡策略，就导致了，第一次访问A服务器，第二次可能访问到B服务器，这个时候存储在A服务器上的session信息在B服务器上读取不到。

4.2 Session管理-Session Sticky粘滞会话：



打个比方就是如果我们每次吃饭都要保证我们用的是自己的碗筷，而只要我们在一家饭店里存着我们的碗筷，只要我们每次去这家饭店吃饭就好了。

对于同一个连接中的数据包，负载均衡会将其转发至后端固定的服务器进行处理。

解决了我们session共享的问题，但是它有什么缺点呢？

* 一台服务器运行的服务挂掉，或者重启，上面的 session 都没了。
* 负载均衡器成了有状态的机器，为以后实现容灾造成了羁绊。

4.3 Session管理-Session 复制



就像我们在所有的饭店里都存一份自己的碗筷。我们随意去哪一家饭店吃饭都OK，不适合做大规模集群，适合机器不多的情况。

解决了我们session共享的问题，但是它有什么缺点呢？

* 应用服务器间带宽问题，因为需要不断同步session数据。
* 大量用户在线时，服务器占用内存过多。

4.4 Session管理-基于Cookie

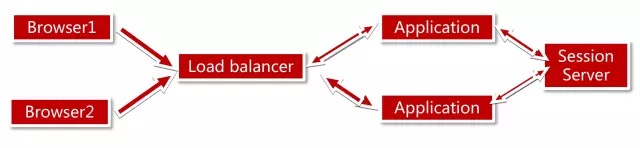


打个比方，就是我们每次去饭店吃饭，都自己带着自己的碗筷。

解决了我们session共享的问题，但是它有什么缺点呢？

* cookie 的长度限制。
* cookie存于浏览器，安全性是一个问题。

4.5 Session管理-Session 服务器



打个比方，就是我们的碗筷都存在了一个庞大的橱柜里，我们去任何一家饭店吃饭，都可以从橱柜中拿到属于我们自己的碗筷。

解决了我们session共享的问题，这种方案需要思考哪些问题呢？

* 保证 session 服务器的可用性，session服务器单点如何解决？
* 我们在写应用时需要做调整存储session的业务逻辑

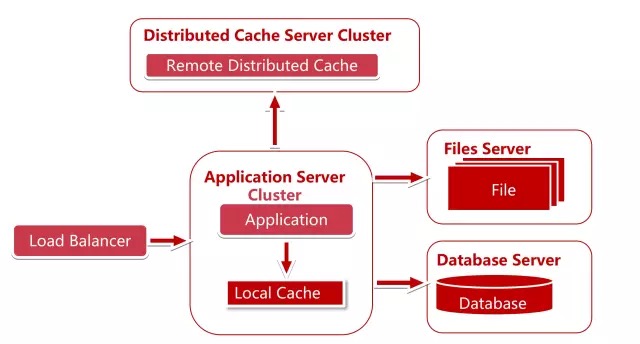
打个比方，我们为了提高session server的可用性，可以继续给session server做集群。

5. 中间总结

所以说，网站架构在遇到某些指标瓶颈时，演进的过程中，都有哪些解决方案，他们都有什么优缺点？业务功能上如何取舍？如何做出选择？这个过程才是最重要的。

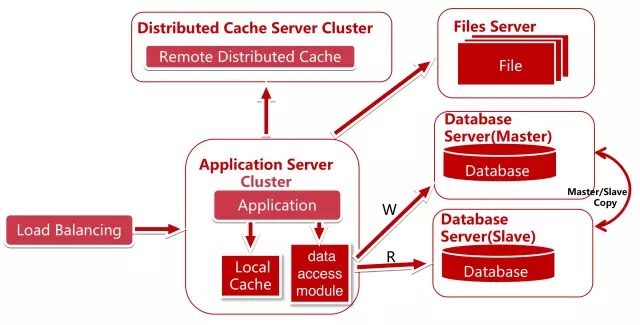
在解决了横向扩展应用服务器之后，那我们继续~~

继续回到目前架构图



数据库的读及写操作都还需要经过数据库。当用户量达到一定量，数据库将会成为瓶颈。那我们如何来解决呢？

6. 数据库读写分离



使用数据库提供的热备功能，将所有的读操作引入slave 服务器，因为数据库的读写分离了，所以，我们的应用程序也得做相应的变化。我们实现一个数据访问模块(图中的data access module)使上层写代码的人不知道读写分离的存在。这样多数据源读写分离就对业务代码没有了侵入。这里就引出了代码层次的演变。

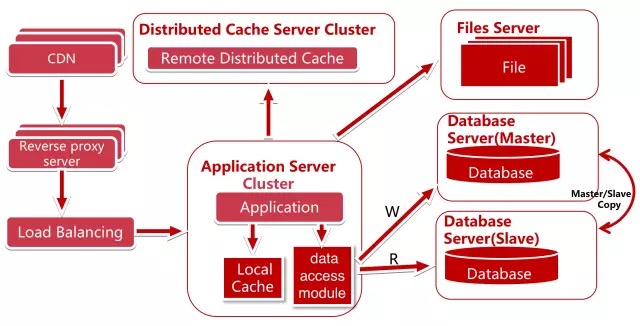
6.1 思考的点

* 如何支持多数据源？
* 如何封装对业务没有侵入？
* 如何使用目前业务的ORM框架完成主从读写分离？是否需要更4. 换ORM模型？ORM模型之间各有什么优缺点？
* 如何取舍？

数据库读写分离会遇到如下问题：

* 在master和slave复制的时候，考虑延时问题、数据库的支持、复制条件的支持。
* 当为了提高可用性，将数据库分机房后，跨机房传输同步数据，这个更是问题。
* 应用对于数据源的路由问题。

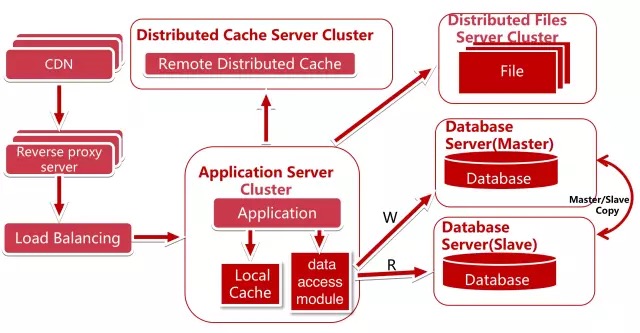
7. 使用反向代理和 CDN 加速网站响应



使用 CDN 可以很好的解决不同的地区的访问速度问题，反向代理则在服务器机房中缓存用户资源。

访问量越来越大，我们文件服务器也出现了瓶颈。

8. 分布式文件系统

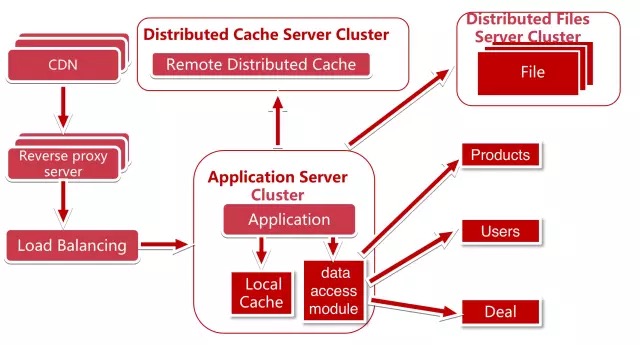


8.1 思考的点

* 分布式文件系统如何不影响已部署在线上的业务访问？不能让某个图片突然访问不到呀
* 是否需要业务部门清洗数据？
* 是否需要重新做域名解析？

这个时候数据库又出现了瓶颈。

9. 数据垂直拆分



数据库专库专用，如图Products、Users、Deal库。

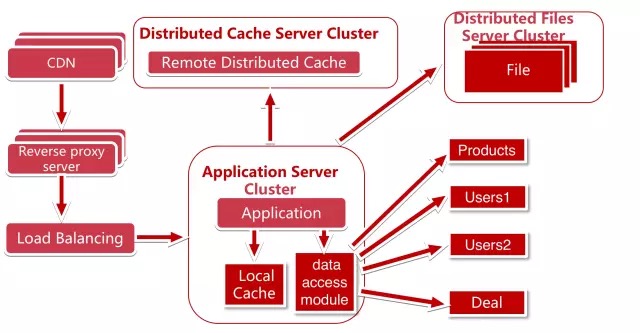
解决写数据时，并发，量大的问题。

9.1 思考的点

* 跨业务的事务？如何解决？使用分布式事务、去掉事务或不追求强事务
* 应用的配置项多了
* 如何跨库进行数据的join操作

这个时候，某个业务的数据表的数据量或者更新量达到了单个数据库的瓶颈。

10. 数据水平拆分



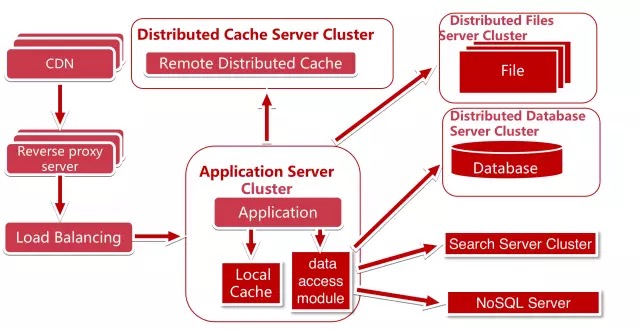
如图，我们把User拆成了User1和User2，将同一个表的数据拆分到两个数据库中，解决了单数据库的瓶颈。

10.1 思考的点

* 水平拆分的策略都有哪些？各有什么优缺点？
* 水平拆分的时候如何清洗数据？
* SQL 的路由问题，需要知道某个 User 在哪个数据库上。
* 主键的策略会有不同。
* 假设我们系统中需要查询2017年4月份已经下单过的用户名的明细，而这些用户分布在user1和user2上，我们后台运营系统在展示时如何分页？

这个时候，公司对外部做了流量导入，我们应用中的搜索量飙升，继续演进。

11. 拆分搜索引擎



使用搜索引擎，解决数据查询问题。部分场景可使用 NoSQL 提高性能，开发数据统一访问模块，解决上层应用开发的数据源问题。如图data access module 可以访问数据库，搜索引擎，NoSQL。

12. 最后总结

这个只是一个举例演示，各个服务的技术架构是需要根据自己业务特点进行优化和演进的，所以大家的过程也不完全相同。

最后的这个也不是完美的，例如负载均衡还是一个单点，也需要集群，我们的这个架构呢也只是冰山一角，沧海一粟。在架构演进的过程中，还要考虑系统的安全性、数据分析、监控、反作弊等等……，同时继续发展呢，SOA架构、服务化、消息队列、任务调度、多机房等等……

从刚才对架构演进的讲解，也可以看出来，所有大型项目的架构和代码，都是这么一步一步的根据实际的业务场景，和发展情况发展演变而来的，在不同的阶段，会使用的不同的技术，不同的架构来解决实际的问题，所以说，高大上的项目技术架构和开发设计实现不是一蹴而就的。

正是所谓的万丈高楼平地起。在架构演进的过程中，小到核心模块代码，大到核心架构，都会不断演进的，这个过程值得我们去深入学习和思考。一起加油吧~~

欢迎大家一起交流~